Splitting forward simulations to cope with liveness

Hesselink, Wim H.

Published in:
Acta informatica

DOI:
10.1007/s00236-006-0007-y

Publication date:
2006

Acta Informatica (2006) 42(8/9): 583–602
DOI 10.1007/s00236-006-0007-y

ORIGINAL ARTICLE

Wim H. Hesselink

Splitting forward simulations to cope with liveness

Received: 22 November 2004 / Revised: 11 January 2006 / Published online: 1 March 2006

Abstract In the literature, the conditions on history variables or forward simulations that are related to liveness are expressed in terms of behaviours, and are stronger than convenient and necessary. In this paper, we propose alternative conditions on the simulation relation, which are expressed in terms of the next state relation, and are closely tied to the weak or strong fairness conditions of the specifications. The proof of soundness of this proposal is based on a new theorem that asserts the existence of a strongly fair scheduler for infinitely many alternatives. The theory is extended to simulations in which the concrete specification (occasionally) does fewer steps than the abstract specification it implements.

1 Introduction

The verification of concurrent algorithms usually splits into two parts: safety (nothing bad happens) and liveness (eventually something good happens). The verification of safety is almost always critical. Indeed, concurrent algorithms for which safety is easy to prove are usually not interesting. The importance of liveness is less predictable: it is sometimes self-evident, sometimes easy, and sometimes difficult. When verification of liveness is difficult, the designer or verifier of the algorithm often feels satisfied with a proof of safety.

The above holds for hand-written proofs in a mathematical style, more formal hand-written proofs in the style advocated by Lamport [15], and also for proofs with proof checkers or theorem provers.

The effect is that the formal theories are less well tested in their liveness aspects than in their safety aspects. One of the central methods for the verification of

W. H. Hesselink (✉)
Department of Mathematics and Computing Science, Rijksuniversiteit Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands
E-mail: wim@cs.rug.nl
URL: http://www.cs.rug.nl/∼wim

concurrent algorithms is the introduction of auxiliary or history variables [1, 19]. Recently, we discovered that the liveness conditions imposed on history variables by [1, 9] are too strong in the sense that they are not satisfied in some cases where addition of a history variable is obviously sound.

These liveness conditions are expressed in terms of behaviours. One of the central aims in concurrency verification is to reduce the role of behaviours in concrete proofs and to concentrate the proof obligations on the next-state relation. This is also a compelling reason to reconsider these conditions.

Finally, even when applicable, these conditions require extensive verification in situations where the human verifier is convinced of the soundness. One of the first rules of mechanical verification is that one should try to eliminate fruitless proof obligations. Thus, inspired by our wish for effective and complete mechanical verification, we had to improve the theory.

We work in the theory of Abadi and Lamport [1], which is based on a form of linear temporal logic. This is a refinement calculus in which programs are concrete specifications that may refine (more) abstract specifications. There are several refinement relations between specifications. In [6, 8], we introduced *simulations* for the general case of trace compatability. The refinement relation corresponding to the addition of history variables is called *forward simulation*. Our aim is to show that the conditions on forward simulations are stronger than convenient and necessary, and to propose *splitting simulations* as alternatives.

In order to prove soundness of splitting simulations, i.e., that they are indeed simulations, we need the theoretical existence of strongly fair schedulers. Technically, we obtained this result many years ago, but we never realized its general abstract form and applicability. Here we present this result in its proper form and with a simpler proof.

It is usually the case that the concrete behaviours do more steps than the abstract behaviours they implement. This is easily incorporated in the formalism by allowing the abstract behaviours to stutter arbitrarily. In [14], Lamport has argued forcefully that formalisms for refinement should also allow concrete behaviours that take (occasionally) fewer steps than the abstract behaviours they implement. Since this adds a technical complication to the theory, we have been reluctant to accept this verdict, but recently we were forced to it by some compelling cases. At this point, we therefore propose to weaken the definition of simulation to something that can be described as trace compatability modulo stutterings. Henceforward, the *simulations* of [6, 8] are called *strict simulations*.

Indeed, in our treatment [7] of the lazy caching algorithm of [2, 13], we need a nonstrict simulation, in which the abstract behaviours must execute some additional actions sufficiently often, i.e., under weak fairness. To handle this, we first tried to use a stuttering version of forward simulations, but the solution turned out to be a nonstrict version of splitting simulations. Actually, we invented the nonstrict *splitting simulations* first, and then saw that strict splitting simulations are also useful. The present paper is thus a companion of the lazy caching paper [7]. Beyond what is needed in [7], we incorporate strong fairness, since we expect that this will be needed in applications to refinement of atomicity.

Overview In Sect. 2, we present our version of specifications and temporal logic. In Sect. 3, we treat strict simulations and forward simulations, and give the

example of an obviously sound history variable that is not justified by a forward simulation. Section 4 is an intermezzo to define strongly fair schedulers and to show their existence.

In Sect. 5, we define splittings of specifications and strict splitting simulations. A specification has a *splitting* iff its supplementary property only consists of weak and strong fairness conditions on alternatives. A strict splitting simulation is a relation between the state spaces of the specifications that respects the alternatives in a certain sense. Nonstrict simulations and splitting simulations are defined and treated in Sect. 6. We conclude in Sect. 7.

Contributions The first point is the observation that the usual progress conditions for history variables are inadequate. The concepts of (strict) splitting simulations and their proofs of soundness are new. As noted by a referee, it is important that the verification of a splitting simulation only requires the next-state relation, whereas the verification of forward simulations requires the analysis of executions and behaviours. It seems that the definitions of fair schedulers and some of the results about them in Sect. 4 are new. All proofs have been verified with the proof assistant PVS [20].

2 Specifications and temporal logic

In this section, we present our formalism for specifications, which follows [1]. Unlike TLA [16], different specifications usually have different state spaces. If X stands for the state space, predicates on X correspond to sets of states, relations over X correspond to possible state transformations, and computations give rise to infinite sequences over X. A specification is a state machine over some state space with a supplementary property to specify progress.

2.1 Predicates and relations

A predicate (boolean function) on a set X can be identified with the subset of X where the predicate holds. We can therefore identify negation ($\neg$) with complementation with respect to X.

A binary relation on a set X is identified with the set of pairs that satisfy the relation; this is a subset of the Cartesian product $X \times X = X^2$. We write $\mathbf{1}$ for the identity relation of X.

2.2 Temporal formulae

Infinite sequences are used to represent consecutive values during computations. We write X^ω for the set of infinite sequences on X, which are regarded as functions $\mathbb{N} \to X$. For a sequence xs, we write $Suf(xs)$ to denote the set of its (infinite) suffixes. If P is a set of sequences, the sets $\Box P$ (always P), and $\Diamond P$ (sometime P) are defined by

$$\begin{aligned} xs \in \Box P &\equiv Suf(xs) \subseteq P, \\ \Diamond P &= \neg \Box \neg P. \end{aligned}$$

So, $xs \in \Box P$ means that all suffixes of xs belong to P, and $xs \in \Diamond P$ means that xs has some suffix that belongs to P.

For $U \subseteq X$, we define the subset $[\![U]\!]$ of X^ω to consist of the sequences whose first element is in U. For a relation A on X, we define the subset $[\![A]\!]_2$ of X^ω to consist of the sequences that start with an A-transition. So we have

$$xs \in [\![U]\!] \equiv xs(0) \in U,$$
$$xs \in [\![A]\!]_2 \equiv (xs(0), xs(1)) \in A.$$

In temporal logic, the operators $[\![\,]\!]$ and $[\![\,]\!]_2$ are usually kept implicit. So, the reader who wants to ignore them is in good company.

A sequence ys is defined to be a *stuttering* of a sequence xs, notation $xs \preceq ys$, iff xs can be obtained from ys by replacing some finite nonempty subsequences ss of consecutive equal elements of ys with their first elements $ss(0)$. For example, if, for a finite list vs, we write vs^ω to denote the sequence obtained by concatenating infinitely many copies of vs, the sequence $(aaabbbccb)^\omega$ is a stuttering of $(abbccb)^\omega$, i.e. $(abbccb)^\omega \preceq (aaabbbccb)^\omega$.

A subset P of X^ω is called a *property* [1, 9] iff it is insensitive to stutterings, i.e., if $P(xs) = P(ys)$ whenever $xs \preceq ys$. If P is a property, then $\Box P$, and $\Diamond P$, and $\neg P$ are properties. The conjunction and disjunction of properties is a property. $[\![U]\!]$ is a property for every $U \subseteq X$. If A is a reflexive relation on X, then $\Box [\![A]\!]_2$ is a property. If A is irreflexive, then $\Diamond [\![A]\!]_2$ is a property.

If X has more than one element, not every subset of X^ω is a property. For example, the set $\Box \Diamond [\![\mathbf{1}]\!]_2$, which consists of the sequences that stutter infinitely often, is not a property. For instance, if $a \neq b$, then $(abb)^\omega \in \Box \Diamond [\![\mathbf{1}]\!]_2$ and $(ab)^\omega \notin \Box \Diamond [\![\mathbf{1}]\!]_2$, while $(ab)^\omega \preceq (abb)^\omega$.

The weak fairness set $WF(A)$ of a relation A is defined to consist of the sequences that take infinitely many A transitions if A is in some suffix always enabled. Following [16], we thus define

$$\begin{aligned} WF(A) &= \Box \Diamond [\![A]\!]_2 \cup \Box \Diamond [\![disabled(A)]\!], \quad \text{where} \\ disabled(A) &= \{x \mid \forall y : (x, y) \notin A\}. \end{aligned}$$

Similarly, the strong fairness set $SF(A)$ is defined to consist of the sequences that take infinitely many A transitions if A is infinitely often enabled:

$$SF(A) = \Box \Diamond [\![A]\!]_2 \cup \Diamond \Box [\![disabled(A)]\!].$$

If A is irreflexive, $WF(A)$ and $SF(A)$ are properties.

2.3 Specifications and programs

Following [1], a *specification* is a tuple $K = (X, Y, N, P)$ where X is the state space, $Y \subseteq X$ is the set of initial states, $N \subseteq X^2$ is the next-state relation and P is the supplementary property. Relation N is required to be reflexive in order to allow stutterings. P is a subset of the set X^ω of the infinite sequences of states, which is required to be a property.

We define an *initial execution* of K to be a sequence xs over X with $xs(0) \in Y$ and such that every pair of consecutive elements belongs to N. A *behaviour* of K

is an infinite initial execution xs of K with $xs \in P$. We write $Beh(K)$ to denote the set of behaviours of K. It is easy to see that

$$Beh(K) = [\![Y]\!] \cap \Box\, [\![N]\!]_2 \cap P.$$

We use specifications to model concurrent systems with shared variables and processes that also have private variables. In this setting, a state of the system is given by the values of all variables, and the state space X is the set of all states.

When convenient, the components of a specification $K = (X, Y, N, P)$ are denoted $states(K) = X$, $start(K) = Y$, $step(K) = N$ and $prop(K) = P$.

Example Let $m \in \mathbb{N}$ be positive. Consider the specification $K(m)$ given by

var $\mathtt{j} : \mathbb{N} := 0$;
do $true \;\; \rightarrow \mathtt{j} := (\mathtt{j} + 1)$ **mod** m **od**;
prop: $\mathtt{j}$ changes infinitely often.

In such a program-like denotation, we keep the stuttering steps implicit. So we have $states(K(m)) = \mathbb{N}$, $start(K(m)) = \{0\}$, relation $step(K(m))$ consists of the pairs (j, k) with $k = (j + 1)$ **mod** m or $j = k$ (stuttering). The supplementary property is $prop(K(m)) = \Box \Diamond [\![\mathtt{j} \neq \mathtt{j}^+]\!]_2$, where $\mathtt{j}^+$ refers to the value of $\mathtt{j}$ in the next state.

Taking $m = 3$, the behaviours of $K(3)$ are the stutterings of $vs = (012)^\omega$. The other initial executions are the stutterings of the infinite sequences $(012)^k 0^\omega$, and $(012)^k 01^\omega$, and $(012)^k 012^\omega$ for $k \in \mathbb{N}$. These are no behaviours since eventually $\mathtt{j}$ is constant in them.

A *visible specification* is a pair (K, v) where K is a specification and v is a function from $states(K)$ to some set of observations. Then v is called the *observation function*. A *visible behaviour* of (K, v) is a sequence vs of observations such that $vs \preceq v \circ xs$ for some behaviour xs.

Example Let $v : \mathbb{N} \rightarrow \mathbb{N}$ be the observation function given by $v(j) = j$ **div** 3. The visible behaviours of $K(15)$ are the stutterings of $vs = (01234)^\omega$. Notice that vs itself is regarded as a visible behaviour even though, for every behaviour xs, the sequence of observations $v \circ xs$ stutters at least twice at every symbol.

3 Implementations and strict simulations

Let (K, v) and (L, w) be visible specifications with observation functions to the same set of observations. Then (K, v) is said to *implement* (L, w) iff every visible behaviour of (K, v) is a visible behaviour of (L, w), see [1].

The easiest way to prove implementation relations between different specifications is by means of refinement mappings. It is well known, however, that refinement mappings are often too specific for this purpose.

Usually, we also need to extend the state space with history variables [1]. Sometimes, we even need prophecy variables [1] or eternity variables [9]. All these methods can be unified as strict simulations, which were introduced in [6, 9].

3.1 Refinement mappings and strict simulations

If K and L are specifications, a function $f : \mathit{states}(K) \to \mathit{states}(L)$ is called a *refinement mapping* [1] from K to L iff $f(x) \in \mathit{start}(L)$ for every $x \in \mathit{start}(K)$, and $(f(x), f(x')) \in \mathit{step}(L)$ for every pair $(x, x') \in \mathit{step}(K)$, and $f \circ xs \in \mathit{prop}(L)$ for every $xs \in \mathit{Beh}(K)$.

The idea of simulation is to generalize a refinement mapping to a binary relation F between $\mathit{states}(K)$ and $\mathit{states}(L)$. For visible specifications (K, v) and (L, w), such a relation F is called *nondisturbing* if F respects the observations in the sense that $v(x) = w(y)$ for all pairs $(x, y) \in F$.

We write F^ω for the relation between infinite sequences given by

$$(xs, ys) \in F^\omega \equiv (\forall\, i : (xs(i), ys(i)) \in F).$$

A relation F between $\mathit{states}(K)$ and $\mathit{states}(L)$ is called a *strict simulation* from specification K to specification L (notation $F : K \twoheadrightarrow L$) if, for every $xs \in \mathit{Beh}(K)$, there exists $ys \in \mathit{Beh}(L)$ with $(xs, ys) \in F^\omega$.

Since every function is a binary relation of a special kind, it is easy to see that, if f is a refinement mapping from K to L, then f is a strict simulation $f : K \twoheadrightarrow L$.

For visible specification (K, v) and (L, w) with some nondisturbing strict simulation $K \twoheadrightarrow L$, it is easy to prove that (K, v) implements (L, w), see [9], Theorem 2.6. We are therefore interested in strict simulations only when they are nondisturbing. The verification whether some relation is nondisturbing, is usually trivial, but it requires explicit observation functions.

In the remainder of this paper, we therefore forget about the observations. Of course, our results are only useful when observations are possible and when the simulations are nondisturbing.

3.2 Forward simulations

The easiest way to prove that one specification simulates another is by starting at the beginning and constructing the corresponding behaviour in the other specification inductively. This idea is formalized in forward simulations [4, 17, 18], defined as follows.

A relation F between $\mathit{states}(K)$ and $\mathit{states}(L)$ is called a *forward simulation* from specification K to specification L iff

(F0) For every $x \in \mathit{start}(K)$, there is $y \in \mathit{start}(L)$ with $(x, y) \in F$.
(F1) For every pair $(x, y) \in F$ and every x' with $(x, x') \in \mathit{step}(K)$, there is y' with $(y, y') \in \mathit{step}(L)$ and $(x', y') \in F$.
(F2) Every initial execution ys of L and every behaviour xs of K, we have that $(xs, ys) \in F^\omega$ implies $ys \in \mathit{prop}(L)$.

The following lemma [9] expresses soundness of forward simulations:

Lemma 1 *Every forward simulation F from a specification K to a specification L is a strict simulation $F : K \twoheadrightarrow L$.*

A forward simulation $F : K \rightharpoonup\!\triangleright L$ is called a *history extension* iff it is the converse of a refinement mapping $L \rightharpoonup\!\triangleright K$. Usually, the state space of K is spanned by some variables, the state space of L is spanned by the same variables together with some auxiliary variables, and the refinement mapping from L to K is the projection function that forgets the values of the auxiliary variables. Roughly speaking, condition (F0) is a matter of consistent initialization, condition (F1) says that the steps of K are faithfully represented by L, and condition (F2) says that no additional progress conditions are imposed.

The conditions (F0) and (F1) go back to [18], but condition (F2) is added in [1]. In almost all applications, safety is of primary importance. Progress is often neglected or treated only informally. Therefore, there is not much experience with condition (F2). Theoretically, condition (F2) is fully justified, since it is strong enough for soundness, i.e. Lemma 1, and weak enough for semantic completeness, cf. [1, 9].

For practical applications, however, it is highly unsatisfactory that condition (F2) is expressed in terms of executions and behaviours. It is preferable to reduce the importance of executions or behaviours, and to rely on the next-state relation as much as possible. A second reason to discard condition (F2), is that it is stronger than convenient. This is shown in the next example.

3.3 A history variable that violates (F2)

The following example is a simplification of a cache updating algorithm. We consider a program with two integer variables $\mathtt{i}$ and $\mathtt{k}$, initially 0, given by

do
A : ‖ $true \rightarrow$ **choose** $\mathtt{i} \in \{j \mid j \neq \mathtt{i} \wedge \mathtt{k} < j\}$;
B : ‖ $\mathtt{i} \leq \mathtt{k} \rightarrow$ **choose** $\mathtt{i} \in \mathbb{Z}$;
C : ‖ $\mathtt{k} < \mathtt{i} \rightarrow \mathtt{k} := \mathtt{i}$;
od;
prop: A and C are treated weakly fair.

Alternative A is always enabled. So, by weak fairness, step A is taken eventually. When A is taken, it establishes $\mathtt{k} < \mathtt{i}$. Steps A and B preserve $\mathtt{k} < \mathtt{i}$. Therefore, by weak fairness, step C is taken eventually, increasing $\mathtt{k}$ and establishing $\mathtt{k} = \mathtt{i}$. Whenever $\mathtt{i} \leq \mathtt{k}$, step B can modify the value of $\mathtt{i}$ arbitrarily. In the cache interpretation, A and B are abstractions of updates of the cache (A being an update on request), while C corresponds to an inspection of the cache.

This program corresponds to a specification K with $\mathit{states}(K) = \mathbb{Z} \times \mathbb{Z}$ and $\mathit{start}(K) = \{(0, 0)\}$ and $\mathit{step}(K) = \mathbf{1} \cup A \cup B \cup C$ and $\mathit{prop}(K) = WF(A) \cap WF(C)$, where A, B, C are regarded as binary relations on $\mathit{states}(K)$ in the natural way. For example, A consists of the pairs of pairs $((i, k), (j, k))$ with $i \neq j \wedge k < j$. Note that A and C are irreflexive, so that $WF(A)$ and $WF(C)$ are properties.

When investigating specification K, it is natural to introduce an integer history variable, say $\mathtt{t}$, to count the number of times alternative A is taken. We take $\mathtt{t} = 0$

initially. We thus compare the above program with

do
A' : ‖ $true \to$ **choose** $\mathtt{i} \in \{j \mid j \neq \mathtt{i} \wedge \mathtt{k} < j\}$; $\mathtt{t} := \mathtt{t} + 1$;
B' : ‖ $\mathtt{i} \leq \mathtt{k} \to$ **choose** $\mathtt{i} \in \mathbb{Z}$;
C' : ‖ $\mathtt{k} < \mathtt{i} \to \mathtt{k} := \mathtt{i}$;
od;
prop: A' and C' are treated weakly fair.

Let L be the corresponding specification, with the state space spanned by $\mathtt{i}$, $\mathtt{k}$, and $\mathtt{t}$. Note that A', B', C' represent binary relations on the state space of L. It is clear that the projection function that deletes the third component of the state forms a refinement mapping $L \rightarrowtriangle K$. Its converse relation $cvf = cv(f)$ should be a history extension $K \rightarrowtriangle L$. Indeed, one easily verifies that cvf satisfies the conditions (F0) and (F1). It is also clear that cvf is a strict simulation: every behaviour of K is easily transferred to L.

Yet, condition (F2) fails. Let ys be the initial execution of L obtained by alternating steps B' with $\mathtt{i} := \mathtt{k} + 1$ and C'. We then have $ys(2n) = (n, n, 0)$ and $ys(2n + 1) = (n + 1, n, 0)$. This execution is not a behaviour of L since step A' is always enabled and never taken. The projection of ys to the state space of K, however, is a behaviour xs of K since the B steps taken are also A steps. This shows that (F2) is violated.

At this point we pay the price for our formalism with unlabelled transitions. The problem does not occur in formalisms where behaviours are sequences of labelled transitions as e.g. [11, 12]. Yet, we prefer unlabelled transitions since we want to avoid the complications of renaming and hiding labels.

The crux of the example is that A and B overlap while A' and B' are disjoint. Condition (F2) fails since relation cvf does not distinguish the alternatives A, B, C, and therefore does not recognize the special relationships between A and A', B and B', and C and C'. We need a remedy that is able to recognize the alternatives. It also turns out that, in case of overlapping alternatives, we need a kind of fair scheduler.

Moreover, in cases where the concrete system occasionally does fewer steps than the abstract system, we need some mechanism to insert abstract steps, possibly under some fairness constraint. This is also a matter of scheduling. In such cases, labelling the transitions would not help but only complicate.

4 Intermezzo: existence of fair schedulers

Fairness conditions and fair schedulers are in some sense opposite sides of the same coin. Fairness conditions are assumptions that constrain the nondeterminacy of specifications. They can sometimes be justified by stochastic or physical considerations. Fair schedulers are deterministic mechanisms that can serve to prove satisfiability of fairness conditions (we come back to this in Sect. 5.1). They are used in this paper since the claim that some relation is a (strict) simulation poses a satisfiability problem.

A scheduler is a deterministic mechanism that can be repeatedly applied to choose between a set of alternatives. To avoid that it always makes the same

choice, the scheduler needs some kind of memory that is updated each time a choice is made. Not all alternatives are always available. We therefore assume that the scheduler's decisions can also be based on a set of enabled alternatives.

The main goal of this section is to introduce strongly fair schedulers and to prove their existence, but we also show that a strongly fair scheduler needs to reckon with the set of enabled alternatives.

4.1 Formalization of schedulers

We let A be the set of alternatives and $P(A)$ be the set of subsets of A. A *scheduler* is a triple (M, c, s) where M is a nonempty set, called the *memory*, and $c : M \times P(A) \to A$ is a function to choose the alternative and $s : M \times P(A) \to M$ is a function to choose the next memory state. The second argument of c and s is regarded as the set of enabled alternatives. Thus, every memory transition may depend on the current set of enabled alternatives.

The scheduler (M, c, s) is called *strict* iff $c(m, p) \in p$ for every m and every nonempty p. Although we strive for strict schedulers, it turns out that non-strict schedulers are also useful since they can easily be made strict. Indeed, if (M, c, s) is a scheduler, we can construct a *strict variation* (M, c', s) of it by defining

$$c'(m, p) = \textbf{if } p = \emptyset \vee c(m, p) \in p \textbf{ then } c(m, p) \\ \textbf{else } \mathit{some\ element\ of}\ p \textbf{ end}.$$

Let us say that an alternative is *taken* when it is chosen in an enabled situation. Choosing an alternative that is not enabled is considered harmless but unproductive.

Repeated application of given scheduler (M, c, s) is modelled as follows. For an initial state $m \in M$ and an infinite sequence of enabling sets $ps \in P(A)^\omega$, we define a sequence $ss(m, ps)$ of memory states inductively by $ss(m, ps)(0) = m$ and $ss(m, ps)(n + 1) = s(ss(m, ps)(n), ps(n))$. The nth choice is given by $cs(m, ps)(n) = c(ss(m, ps)(n), ps(n))$. In this way, we obtain the infinite sequence $cs(m, ps) \in A^\omega$, which represents the sequence of chosen alternatives, based on the memory initialization m and the invocations with $ps(n)$ as consecutive enabling sets.

The fairness quality of the scheduler depends on the answer to the following question. If alternative $a \in A$ is enabled sufficiently often, will it be taken sufficiently often? More precisely, if a occurs sufficiently often in a sequence of enabling sets ps, will there be sufficiently many indices n with $cs(m, ps)(n) = a \in ps(n)$? Depending on the interpretation of the words "sufficiently," we thus obtain the following concepts of weakly and strongly fair schedulers.

The scheduler (M, c, s) is called *weakly fair* iff every alternative that is eventually always enabled, is taken infinitely often. This is formalized in

$$WF: \quad \forall\, ps, m, a, n : \quad (\forall\, k : n \le k \Rightarrow a \in ps(k)) \\ \Rightarrow \quad (\exists k : n \le k \wedge cs(m, ps)(k) = a \in ps(k)).$$

The scheduler (M, c, s) is called *strongly fair* iff every alternative that is enabled infinitely often, is taken infinitely often. This is formalized in

$$SF: \ \forall\, ps, m, a: \quad (\forall\, n: \exists k: n \le k \wedge a \in ps(k)) \\ \Rightarrow \quad (\forall\, n: \exists k: n \le k \wedge cs(m, ps)(k) = a \in ps(k)).$$

It is straightforward to prove:

Lemma 2 *(a) Every strongly fair scheduler is weakly fair.*
(b) If (M, c, s) is weakly or strongly fair, then every strict variation (M, c', s) of it has the same property.
(c) Let (M, c, s) be a scheduler such that the set $\{n \mid cs(m, ps)(n) = a\}$ is infinite for every m, ps, and a. Then (M, c, s) is weakly fair.

Example An infinite "round robin" scheduler for $A = \mathbb{N}$.

Let M consist of the pairs (i, j) with $i \le j$. The scheduler (M, c, s) is given by

$$c((i, j), p) = i,$$
$$s((i, j), p) = \textbf{if} \quad i < j \quad \textbf{then} \quad (i + 1, j) \quad \textbf{else} \quad (0, j + 1) \quad \textbf{end}.$$

This scheduler ignores its argument p and is therefore not strict. It can be described as going round and round in ever growing circles. Every natural number is chosen infinitely often. Therefore, Lemma 2(c) implies that the scheduler is weakly fair. Since it can be made strict easily, we regard it as a useful scheduler.

In the above scheduler, the functions c and s ignore their second arguments. It is easy to show that, in any weakly fair scheduler, the functions c and s do not ignore their first arguments, if A has at least two elements.

We introduced the second argument for the functions c and s in order to enable strong scheduling. As the next result shows, this is indeed necessary.

Lemma 3 *Assume A has at least two elements. Let (M, c, s) be a strongly fair scheduler. Then both functions c and s do not ignore their second arguments.*

Proof Choose $m \in M$, and choose alternatives $a \neq b$ in A.

First assume that function c ignores its second argument. Let M_b be the set of memory states m' with $c(m', A) = b$ (the choice of argument A is arbitrary, since c ignores its second argument). By mutual recursion, we define a sequence of memory states ms and a sequence of enabling sets ps by

$$ms(0) = m,$$
$$ps(n) = \textbf{if} \ \ ms(n) \in M_b \ \textbf{then} \ \{a\} \ \ \textbf{else} \ \{a, b\} \ \textbf{end},$$
$$ms(n + 1) = s(ms(n), ps(n)).$$

Since the scheduler is strongly fair and alternative a is always enabled by ps, alternative a is taken infinitely often.

On the other hand, by induction, we have $ms(n) = ss(m, ps)(n)$ for all n. Since function c ignores its second argument, it follows that, for every index n, we have $cs(m, ps)(n) = c(ms(n), ps(n)) = c(ms(n), A)$. By the definitions of M_b and ps, this implies that, for all n,

$$(*) \quad cs(m, ps)(n) = b \equiv ms(n) \in M_b \equiv b \notin ps(n).$$

So, in this case, alternative b is never chosen when enabled. Since the scheduler is strongly fair, this implies that for some n we have

$$\forall k : n \leq k \Rightarrow b \notin ps(k).$$

Using (*), we get that $cs(m, ps)(k) = b$ for all $k \geq n$. This contradicts the fact that alternative a is taken infinitely often.

Secondly, assume that function s ignores its second argument. Consider the sequence of enabling sets $qs \in P(A)^\omega$ given by $qs(n) = \{a, b\}$. Consider the sequence of memory states mt given by $mt = ss(m, qs)$. Since function s ignores its second argument, we have, for every sequence $rs \in P(A)^\omega$, that $ss(m, rs) = mt$ and hence $cs(m, rs)(n) = c(mt(n), rs(n))$.

Alternative a is infinitely often enabled by qs. Since (M, c, s) is strongly fair, it follows that the set $U = \{n \in \mathbb{N} \mid c(mt(n), \{a, b\}) = a\}$ is infinite. Now consider the sequence of enabling sets rs given by

$$rs(n) = \textbf{if } n \in U \textbf{ then } \{a, b\} \textbf{ else } \{a\} \textbf{ end}.$$

Since U is infinite, alternative b is infinitely often enabled by rs. Since (M, c, s) is strongly fair, there are infinitely many indices n with $cs(m, rs)(n) = b \in rs(n)$. These indices clearly satisfy $n \in U$ and hence $cs(m, rs)(n) = c(mt(n), rs(n)) = c(mt(n), \{a, b\}) = a$. This is a contradiction. □

Like other liveness conditions, strong fairness of a scheduler can be verified by means of a variant function. Let (M, c, s) be a scheduler. Let $(W, \leq)$ be a well-founded partially ordered set. Recall that this implies that every descending sequence in W is eventually constant.

Consider for a function $vf : M \times A \to W$ and elements $m \in M$, $p \in P(A)$, $a \in A$ the conditions

(D0) $vf(s(m, p), a) \leq vf(m, a) \vee c(m, p) = a \in p$,
(D1) $c(m, p) \neq a \in p \Rightarrow vf(s(m, p), a) \neq vf(m, a)$.

Condition (D0) means that, at alternative a, function vf descends unless a is taken. Together with (D0), condition (D1) implies that vf decreases at a whenever a is enabled and not taken.

Lemma 4 *Let the scheduler (M, c, s) have a function vf that satisfies conditions (D0) and (D1) for all m, p, a. Then it is strongly fair.*

Proof Let $m \in M$ be a given initial memory state, and let ps be a sequence of enabling sets. Let ms be the sequence of successive memory states given by $ms = ss(m, ps)$. Assume that b is an alternative that is infinitely often enabled by ps, but is not taken after time t_0. By condition (D0), the sequence $rs = \lambda n : vf(ms(n), b)$ in W is descending after t_0. Since $(W, \leq)$ is well-founded, there is $t_1 \geq t_0$ such that rs is constant beyond t_1. Since b is enabled infinitely often, there is $t_2 \geq t_1$ with $b \in ps(t_2)$. Since b is not taken in t_2, condition (D1) implies that $vf(ms(t_2 + 1), b) \neq vf(ms(t_2), b)$, that is $rs(t_2 + 1) \neq rs(t_2)$. This is a contradiction. □

4.2 Constructing a strongly fair scheduler

It seems that the first construction of a strict and strongly fair scheduler for a finite set A is due to Dijkstra [3]. In [5], we generalized this construction to the case $A = \mathbb{N}$. Unfortunately, that construction is rather nasty. In order to appreciate the problem for infinite A, let us first sketch a different solution for finite A.

If A is a finite set or, more generally, if the set of enabled alternatives is always finite, one can construct a strict and strongly fair scheduler in the following way. The memory M consists of a FIFO queue of alternatives, which is initially empty. If there is an enabled alternative in the queue, function c chooses the first enabled alternative in the queue, and function s removes this alternative from the queue and adds all enabled alternatives that are not yet in the queue at the end of the queue. If there are enabled alternatives but no element of the queue is enabled, c chooses some enabled alternative and s adds all other enabled alternatives at the end of the queue. If there are no enabled alternatives, c chooses an arbitrary alternative and s leaves the queue unchanged. Note that the queue remains finite since the sets of enabled alternatives are finite. It follows that, whenever an alternative is enabled and not taken, it enters the queue or moves towards the front of the queue. This implies strong fairness.

When the sets of enabled alternatives can be infinite, the queue becomes infinite and the above construction fails since there is no end to place enabled alternatives. The following construction is simpler than the one of [5]. It is based on a suggestion by J. E. Jonker. The idea is to put all alternatives in a queue, to always choose the first enabled alternative from the queue, and to move an alternative that is taken, backward in the queue to a position determined by the time.

Theorem 1 *Assume $A = \mathbb{N}$. Then there exists a strict and strongly fair scheduler.*

Proof Although $A = \mathbb{N}$, we still use A to indicate the set of numbers that serve as alternatives. We construct a scheduler (M, c, s) by taking $M = H \times \mathbb{N}$, where H is the set of functions $h : A \to \mathbb{N}$ with $\lim_{n\to\infty} h(n) = \infty$. This implies that, for every $k \in \mathbb{N}$, the set $\{a \mid h(a) \leq k\}$ is finite. For a pair $(h, t) \in M$, function h gives the *load* of the alternatives and component t gives the *time*. The choice function always chooses an alternative a with the lowest load $h(a)$, the successor function updates function h and increments the time. More precisely, we use function h to define the relation $\sqsubseteq_h$ on A given by

$$a \sqsubseteq_h b \equiv h(a) \leq h(b) \wedge (h(a) < h(b) \vee a \leq b).$$

This is the lexical ordering on the pairs $(h(a), a)$. Therefore, relation $\sqsubseteq_h$ is a linear order on A. Clearly, for every $b \in A$, its set of predecessors $Pred(h, b) = \{a \in A \mid a \sqsubseteq_h b\}$ satisfies $Pred(h, b) \subseteq \{a \mid h(a) \leq h(b)\}$ and is therefore finite because of $h \in H$. It follows that every nonempty subset p of A has an element $Min(h, p) \in p$ with $Min(h, p) \sqsubseteq_h a$ for all $a \in p$. The choice function c is now defined by $c((h, t), p) = C(h, p)$ where

$$C(h, p) = \textbf{if } p = \emptyset \textbf{ then } 0 \textbf{ else } Min(h, p) \textbf{ end}.$$

The successor function s of the scheduler is defined by

$$\begin{aligned} s((h, t), p) &= (h', t + 1) \text{ where} \\ h'(a) &= \textbf{if } c((h, t), p) = a \in p \wedge h(a) \leq t \textbf{ then } t + 1 \\ &\quad \textbf{else } h(a) \textbf{ end}. \end{aligned}$$

So, s increments the time and modifies h only at a when a is taken and the load is not higher than the time. Then the new value of the load is the new time. Note that since h grows to infinity, function h' also grows to infinity, so that, indeed s is a function $M \times P(A) \to M$.

Since M is nonempty, the triple (M, c, s) is a scheduler. It follows from the definition of function c that the scheduler (M, c, s) is strict.

We use Lemma 4 to prove strong fairness. To show that alternative a, when enabled often enough, is eventually taken, we first show that, if a is not taken, t grows until $h(a) \leq t$, and when this holds, the number of predecessors of a decreases until a itself is taken. This is formalized by constructing a variant function that satisfies (D0) and (D1). To measure the growth of t towards the load, we define $vf0((h, t), a) = \max(0, h(a) - t)$ and verify, for all $m = (h, t) \in M$,

(0) $vf0(s(m, p), a) \leq vf0(m, a)$,
(1) $t < h(a) \Rightarrow vf0(s(m, p), a) < vf0(m, a)$.

In order to formalize the second part of the argument, we write $s_1(h, t, p)$ for the first component of $s((h, t), p)$. Inspired by condition (D0), we observe that

(2) $Pred(s_1(h, t, p), a) \subseteq Pred(h, a) \vee C(h, p) = a \in p$.

In view of (D1), we verify that

(3) $h(a) \leq t \wedge C(h, p) \neq a \in p \Rightarrow C(h, p) \in Pred(h, a) \setminus Pred(s_1(h, t, p), a)$.

We define $vf1(h, a)$ as the number of elements of the finite set $Pred(h, a)$. The observations (2) and (3) imply

(4) $vf1(s_1(h, t, p), a) \leq vf1(h, a) \vee C(h, p) = a \in p$,
(5) $h(a) \leq t \wedge C(h, p) \neq a \in p \Rightarrow vf1(s_1(h, t, p), a) < vf1(h, a)$.

Let $vf : M \times A \to \mathbb{N}$ be defined by

$$vf((h, t), a) = vf0((h, t), a) + vf1(h, a).$$

The formulae (0) and (4) imply that vf satisfies condition (D0). Since $C(h, p) \neq a \in p$ implies that we do not have $C(h, p) = a \in p$, the formulae (0), (1), (4), and (5) yield (D1). Since $\mathbb{N}$ is well-founded, this concludes the proof. □

Remark In Sect. 5 of [5], we did not yet have the concept of scheduler, but we essentially constructed a strongly fair scheduler (M, c, s) with $M \subseteq (A \to \mathbb{N})$ for which one can use a variant function vf with $vf(m, a) = m(a)$. In that construction, however, the set M and the function s are quite complicated.

Above, we can restrict H to the set of the functions $h : \mathbb{N} \to \mathbb{N}$ for which the set $\{a \mid h(a) \neq a\}$ is finite. This is a countable set. Then $M = H \times \mathbb{N}$ is also countable.

5 Splittings

Inspired by Sect. 3.3, we propose a formalism to introduce alternatives in specifications and to impose weak or strong fairness conditions for these alternatives.

A *splitting* of a specification K consists of a family of subrelations of the next-state relation $step(K)$. These subrelations, to be called the *alternatives*, are

divided in three classes: the weak alternatives, the strong alternatives, and one unfair alternative. It is assumed that the supplementary property of the specification is equivalent to the condition that the weak alternatives are treated under weak fairness and the strong alternatives under strong fairness.

For example, the specification may represent a system of n processes that each read their message boxes with weak fairness, write results when they have them with strong fairness, listen to interrupts with strong fairness, and also do some other things without fairness constraints. In that case, there are n weak alternatives and $2n$ strong alternatives. When the specification allows process creation, however, one must reckon with unboundedly many alternatives, so that the (static) specification must allow infinitely many of them.

When we need to compare the specification K with another specification L, we use a relation between the state spaces of K and L. This relation will be a splitting simulation if both K and L have splittings such that the weak/strong alternatives of K correspond to the weak/strong alternatives of L, that every step according to some alternative in K can be transferred to the corresponding alternative in L, and that for every pair of related states the disabled alternatives for K are also disabled for L.

5.1 The splitting format

We now choose a format to present the weak alternatives and the strong alternatives on an equal footing. We want to allow as many of them as possible. It is harmless to introduce additional empty alternatives since all sequences of states belong to $WF(\emptyset)$ and $SF(\emptyset)$. We therefore allow infinitely many alternatives. We put the unfair alternative at index 0 and use an arbitrary set wf of positive natural numbers to specify the weak alternatives. These considerations lead to the following two definitions.

If wf is a set of positive natural numbers, a *wf-splitting* of specification K is defined as a family of relations $(i \in \mathbb{N} : A.i)$ such that

$$\text{(S0)} \qquad step(K) = \mathbf{1} \cup (\cup\, i \in \mathbb{N} :\ A.i),$$

$$\text{(S1)} \qquad prop(K) = (\cap\, i \in \mathbb{N}_+ : \textbf{if } i \in wf \textbf{ then } WF(A.i) \textbf{ else } SF(A.i)).$$

So, every nonstuttering step of K belongs to some alternative $A.i$. The alternatives in wf are treated with weak fairness, the other positive alternatives are treated with strong fairness. There is no fairness condition for the unfair alternative $A.0$.

If K is a specification with a wf-splitting, a strongly fair scheduler (M, c, s) can be used to implement the fairness requirements of K, i.e., one can use the scheduler to construct a specification L with a very weak supplementary property, together with a refinement mapping $L \twoheadrightarrow K$. One uses $states(K) \times M$ as the state space of L. In the next-state relation of L, the scheduler chooses each time an appropriate alternative from the splitting. Since this construction is not needed for the theory of splitting forward simulations, we leave the details to the interested reader.

5.2 Strict splitting simulations

Let K and L be specifications. A *strict splitting simulation* from K to L is a relation F between the state spaces of K and L such that condition (F0) of Sect. 3.2 holds and that there exist a subset $wf \subseteq \mathbb{N}_+$ and wf-splittings $(i \in \mathbb{N} : A.i)$ and $(i \in \mathbb{N} : B.i)$ of K and L, respectively, that satisfy

(F1s) If $(x, y) \in F$ and $(x, z) \in A.i$, there is w with $(z, w) \in F$ and $(y, w) \in B.i$.
(F2s) If $(x, y) \in F$ and $i > 0$ and $x \in disabled(A.i)$, then $y \in disabled(B.i)$.

Note that condition (F1) of Sect. 3.2 is represented by the conjunction of the formulae (F1s) for all i. Also, note that condition (F2s) for $i = 0$ is vacuous. Soundness of strict splitting simulations is expressed by

Theorem 2 *Every strict splitting simulation is a strict simulation.*

Before proving this theorem, we need to discuss and mollify some of the complicating factors. Firstly, the proof is easy when one imposes the additional assumption that the alternatives $A.i$ with $i > 0$ are pairwise disjoint. In view of Sect. 3.3, however, we do not want to make this assumption. When the alternatives can overlap, we need a scheduler to choose fairly between the alternatives. This is the reason for the intermezzo in Sect. 4.

A second complication is the appearance of $\mathbf{1}$ in condition (S0) of splitting. It is put there since it is useful in almost all applications of the theorem. On the other hand, it complicates the proof considerably. We therefore define a family $(i \in \mathbb{N} : A.i)$ to be a *full* wf-splitting if condition (S0) is replaced by

(S0′) $step(K) = (\bigcup i \in \mathbb{N} : A.i)$.

Every full splitting is a splitting because of reflexivity of $step(K)$. Conversely, if $(i \in \mathbb{N} : A.i)$ is a splitting of specification K, the family $(i \in \mathbb{N} : A'.i)$ defined by $A'.0 = \mathbf{1} \cup A.0$ and $A'.i = A.i$ for $i > 0$, is easily seen to be a full splitting of K.

Proof of the theorem. Let F be a strict splitting simulation from specification K to specification L. So, there exist a subset $wf \subseteq \mathbb{N}_+$ and wf-splittings $(i \in \mathbb{N} : A.i)$ and $(i \in \mathbb{N} : B.i)$ of K and L, respectively, such that the conditions (F0), (F1s) and (F2s) hold. We can replace the splittings by full splittings. This only threatens condition (F1s) for $i = 0$ because of $A'.0 = \mathbf{1} \cup A.0$. The condition remains valid since, if $z = x$, one can choose $w = y$ in (F1s). This shows that we may assume that both splittings are full.

In order to prove that F is a strict simulation $K \rightarrowtriangle L$, we need to transfer an arbitrary behaviour xs from K to L. So, let $xs \in Beh(K)$. We need to construct $ys \in Beh(L)$ with $(xs, ys) \in F^\omega$.

We now apply Theorem 5 with $A = \mathbb{N}$, the index set of the wf-splitting to obtain some strict and strongly fair scheduler (M, c, s). Since we need to choose alternatives i with $(xs(n), xs(n + 1)) \in A.i$, we define the sequence of enabling sets $ps \in P(\mathbb{N})^\omega$ by

$$ps(n) = \{i \mid (xs(n), xs(n + 1)) \in A.i\}.$$

Note that, since xs is a behaviour and $\mathit{step}(K) = (\bigcup i \in \mathbb{N} : A.i)$, we have that $ps(n)$ is always nonempty. Since the scheduler is strict, it follows that $cs(m, ps)(n) \in ps(n)$ for all n.

Since (M, c, s) is a scheduler, M is nonempty and we can choose an initial state $m_0 \in M$. Because of condition (F0), we can choose $y_0 \in \mathit{start}(L)$ with $(xs(0), y_0) \in F$. We now construct a sequence ys in $states(L)$ inductively. First, take $ys(0) = y_0$.

If $y = ys(n)$ has been chosen with $(xs(n), y) \in F$, put $i = cs(m_0, ps)(n)$. Then we have $i \in ps(n)$ and hence $(xs(n), xs(n+1)) \in A.i$. We can therefore use condition (F1s) to choose an element y' with $(xs(n+1), y') \in F$ and $(y, y') \in B.i$. Then define $ys(n+1) = y'$. This constructs an infinite sequence ys in $states(L)$ with $(xs, ys) \in F^\omega$.

We have $ys(0) \in \mathit{start}(L)$. For every n, we have $(ys(n), ys(n+1)) \in B.i$ for some i. Since $\mathit{step}(L) = (\bigcup i : B.i)$, this implies that ys is an initial execution of L. It remains to prove that ys satisfies the supplementary property of L.

Let $i > 0$ be given. We need to prove $ys \in WF(B.i)$ if $i \in wf$ and $ys \in SF(B.i)$ if $i \notin wf$. Since xs is a behaviour of K, we have $xs \in WF(A.i)$ if $i \in wf$ and $xs \in SF(A.i)$ if $i \notin wf$. Since $(xs, ys) \in F^\omega$, condition (F2s) implies

$$xs \in \Box\Diamond [\![disabled(A.i)]\!] \Rightarrow ys \in \Box\Diamond [\![disabled(B.i)]\!],$$
$$xs \in \Diamond\Box [\![disabled(A.i)]\!] \Rightarrow ys \in \Diamond\Box [\![disabled(B.i)]\!].$$

In view of the definitions of WF and SF, it therefore remains to prove

$$xs \in \Box\Diamond [\![A.i]\!]_2 \Rightarrow ys \in \Box\Diamond [\![B.i]\!]_2.$$

This is done by contraposition. Assume $ys \notin \Box\Diamond [\![B.i]\!]_2$. Then there is n_0 with $(ys(r), ys(r+1)) \notin B.i$ for all $r \geq n_0$. By the construction of ys, this implies $i \neq cs(m_0, ps)(r)$ for all $r \geq n_0$. Since (M, c, s) is a strongly fair scheduler, this implies the existence of $n_1 \geq n_0$ with $i \notin ps(r)$ for all $r \geq n_1$. This means $(xs(r), xs(r+1)) \notin A.i$ for all $r \geq n_1$. It follows that $xs \notin \Box\Diamond [\![A.i]\!]_2$. This concludes the proof. □

Now that we have soundness, we can discuss methodology and compare forward simulations with strict splitting simulations. Firstly, strict splitting simulations can only be used when a splitting is available, but that is often the case. When a natural splitting is available, verification of condition (F1s) is usually almost the same as verification of (F1).

Verification of (F2s), however, is usually much easier than verification of (F2). Indeed, (F2s) is a condition on the disabledness of the alternative step relations, whereas (F2) requires an analysis of behaviours. For example, it is easy to see that relation cvf of Sect. 3.3 is a strict splitting simulation.

6 Nonstrict simulations

As Lamport [14] has argued, it is important to allow refinement relations where the concrete behaviour occasionally takes fewer steps than the abstract behaviour. We do this by defining (nonstrict) simulations, which indeed admit additional stutterings in the concrete specification.

At this point, we have to give a more formal definition of the stuttering relation $\preceq$ introduced in Sect. 2.2. We define a function $g : \mathbb{N} \to \mathbb{N}$ to be a *stutter function* iff it is surjective and monotonic. Equivalently, function g is a stutter function iff $g(0) = 0$ and g is unbounded, and $g(i+1) - g(i) \in \{0, 1\}$ for all $i \in \mathbb{N}$. We now define $xs \preceq xt$ to mean that there is a stutter function g with $xs \circ g = xt$. The reader who is in doubt about the direction may note that every stuttering of g induces a stuttering of $xs \circ g$, even if xs is stutter-free.

It is easy to see that the definition implies that $\preceq$ is reflexive and transitive. One can also prove that $\preceq$ is antisymmetric.

A relation F between the state spaces of specifications K and L is defined to be a *simulation* [10] from K to L, notation $F : K \rightarrowtriangle\!\!\!\!\rightarrow L$, if for every $xs \in Beh(K)$ there exists a pair $(xt, ys) \in F^\omega$ with $xs \preceq xt$ and $ys \in Beh(L)$. Sequence xt is a behaviour of K obtained from xs by adding stutterings, in such a way that it matches ys via relation F.

It is easy to see that every strict simulation is a simulation since one can choose $xt = xs$. Conversely, not all simulations are strict. In [10] it is proved that a visible specification (K, v) implements (L, w) if and only if there is a nondisturbing simulation $K \rightarrowtriangle\!\!\!\!\rightarrow L$. In the next section, we describe a class of nonstrict simulations. The stuttering variables of [13] form another example of a nonstrict simulation.

6.1 Splitting simulations

We now define splitting simulations as a mild weakening of the strict splitting simulations of Sect. 5. We only weaken condition (F2s) by adding a stuttering possibility.

A *splitting simulation* from K to L is defined to be a relation F between the state spaces of K and L such that condition (F0) of Sect. 3.2 holds and there is a subset $wf \subseteq \mathbb{N}_+$ such that K and L have wf-splittings $(i \in \mathbb{N} : A.i)$ and $(i \in \mathbb{N} : B.i)$, respectively, such that condition (F1s) holds and:

(F2ns) If $(x, y) \in F$ and $i > 0$ and $x \in disabled(A.i)$, then $y \in disabled(B.i)$ or there exists w with $(y, w) \in B.i$ and $(x, w) \in F$.

Clearly, condition (F2s) of Sect. 5 implies condition (F2ns). There are two principal possibilities to satisfy condition (F2ns). Let alternative i be called *conservative* iff, for every pair $(x, y) \in F$, we have that $x \in disabled(A.i)$ implies $y \in disabled(B.i)$ as in (F2s). Let alternative i be called *stuttering* iff, for every pair $(x, y) \in F$ and every w with $(y, w) \in B.i$, we have that $(x, w) \in F$. It is easy to see that (F2ns) holds if every alternative i is conservative or stuttering. Note that (F2ns) can hold while $A.i$ is empty (i.e. absent) and $B.i$ is nonempty.

Soundness of splitting simulations is expressed by

Theorem 7 *Every splitting simulation is a simulation.*

Proof Let F be a splitting simulation from K to L. By an argument completely analogous to the one used in the beginning of the proof of Theorem 2, we may assume that K and L have full wf-splittings $(i \in \mathbb{N} : A.i)$ and $(i \in \mathbb{N} : B.i)$ satisfying (F1s) and (F2ns).

Let xs be a behaviour of K. We have to construct a behaviour ys of L and a stutter function g with $(xs \circ g, ys) \in F^\omega$. We first use condition (F0) to choose a state $y_0 \in \mathit{start}(L)$ with $(xs(0), y_0) \in F$. We use Theorem 1 to choose a strict and strongly fair scheduler (M, c, s) and a start state $m_0 \in M$.

For $y \in \mathit{states}(L)$ and $k \in \mathbb{N}$, we define the set $pp(y, k)$ of alternatives by

$$\begin{aligned} &i \in pp(y,k) \\ &\quad \equiv (xs(k), xs(k+1)) \in A.i \vee (\exists w : (y,w) \in B.i \wedge (xs(k), w) \in F). \end{aligned}$$

We use simultaneous recursion to construct three infinite sequences $ys \in \mathit{states}(L)^\omega$, $ks \in \mathbb{N}^\omega$, and $ms \in M^\omega$. The start is $ys(0) = y_0$ and $ks(0) = 0$ and $ms(0) = m_0$. Note that $(xs(0), y_0) \in F$. Assume that $y = ys(n)$, $k = ks(n)$, and $m = ms(n)$ have been constructed and satisfy $(xs(k), y) \in F$.

Since xs is a behaviour of K and $(i \in \mathbb{N} : A.i)$ is a full splitting of K, there is an alternative j with $(xs(k), xs(k+1)) \in A.j$. Therefore, $pp(y, k)$ is nonempty. Since the scheduler is strict, it follows that the alternative $i = c(pp(y, k), m)$ satisfies $i \in pp(y, k)$. We use this alternative i to define $ks(n+1) = k'$, $ms(n+1) = m'$, and to choose $ys(n+1) = y'$ by the clauses:

$$\begin{aligned} k' &= \textbf{if } (xs(k), xs(k+1)) \in A.i \textbf{ then } k+1 \textbf{ else } k \textbf{ end}, \\ m' &= s(i, m), \\ y' &\in \{w \mid (y,w) \in B.i \wedge (xs(k'), w) \in F\}. \end{aligned}$$

We have to argue the existence of y' with the properties claimed. If $k' = k+1$, this follows from condition (F1s). If $k' = k$, it follows from $i \in pp(y, k)$ and the definition of k'. The definition of y' implies that indeed $(xs(ks(n+1)), ys(n+1)) \in F$. The resulting sequence ys is an execution of L, since $ys(0)$ is a start state, and every step $(ys(n), ys(n+1))$ belongs to some relation $B.i \subseteq \mathit{step}(L)$.

The sequence ks starts at 0 and takes steps of 0 or 1. Therefore, in order to show that ks is a stutter function, we only need to show that it tends to infinity. If it does not tend to infinity, it eventually becomes constant, say $ks(n) = k_0$ for all $n \geq n_0$. Since $\mathit{step}(K) = (\bigcup i : A.i)$ and xs is a behaviour, there is an alternative i_0 with $(xs(k_0), xs(k_0+1)) \in A.i_0$. Let the sequence of sets ps be given by $ps(n) = pp(ys(n), ks(n))$. Then we have $i_0 \in ps(n)$ for all $n \geq n_0$. Since the scheduler is strongly fair, it follows that there is an index $n \geq n_0$ with $cs(ps)(n) = i_0$. The corresponding step satisfies $k' = k+1$, contradicting the assumption. This proves that ks is a stutter function with $(xs \circ ks, ys) \in F^\omega$.

It remains to prove that ys satisfies the supplementary property of L. In view of the definition of splittings, it suffices to prove for all $i > 0$ that $xs \in WF(A.i)$ implies $ys \in WF(B.i)$ (for $i \in wf$) and that $xs \in SF(A.i)$ implies $ys \in SF(B.i)$ (for $i \notin wf$). Let $i > 0$ be given.

If $ys \in \Box\Diamond[\![B.i]\!]_2$ then ys belongs to both $WF(B.i)$ and $SF(B.i)$. So, we may assume that $ys \notin \Box\Diamond[\![B.i]\!]_2$. This implies the existence of n_1 such that $(ys(r), ys(r+1)) \notin B.i$ for all $r \geq n_1$. By the construction of ys, it follows that $i \neq cs(ps)(r)$ for all $r \geq n_1$. Since the scheduler is strongly fair, this implies that there is $n_2 \geq n_1$ such that $i \notin ps(r)$ for all $r \geq n_2$. By the definitions of ps and pp this implies

$$\begin{aligned} (**) \qquad &\forall r : n_2 \leq r \Rightarrow (xs(ks(r), xs(ks(r)+1)) \notin A.i \\ &\qquad\qquad \wedge \quad \neg(\exists w : (ys(r), w) \in B.i \wedge (xs(ks(r)), w) \in F). \end{aligned}$$

Since ks is a stutter function, the first conjunct of (**) yields $xs \notin \Box\Diamond[\![A.i]\!]_2$.

On the other hand, since $(xs(ks(r)), ys(r)) \in F$, condition (F2ns) together with the second conjunct of formula (**) implies

$$\forall r : n_2 \leq r \wedge xs(ks(r)) \in disabled(A.i) \Rightarrow ys(r) \in disabled(B.i).$$

Using that ks is a stutter function, it follows that

$$xs \in \Box\Diamond[\![disabled(A.i)]\!] \Rightarrow ys \in \Box\Diamond[\![disabled(B.i)]\!],$$
$$xs \in \Diamond\Box[\![disabled(A.i)]\!] \Rightarrow ys \in \Diamond\Box[\![disabled(B.i)]\!].$$

Since $xs \notin \Box\Diamond[\![A.i]\!]_2$, it follows that $xs \in WF(A.i)$ implies $ys \in WF(B.i)$, and $xs \in SF(A.i)$ implies $ys \in SF(B.i)$. This concludes the proof. □

7 Conclusion

For actual correctness proofs in which liveness is not neglected, (strict) splitting simulations form a more convenient tool than the classical forward simulations, since they only require investigation of the next-state relations and do not generate conditions on the behaviours. Of course, they can only be applied to prove simulation relations between specifications with related supplementary properties given in terms of WF and SF.

The theory was primarily developed for the application [7] of splitting simulations to the lazy caching algorithm of [2]. We have not yet worked on other applications of the theory. It came as a surprise to formalize and reuse old work on a fair scheduler for infinitely many alternatives.

At several points during the development of the theory, the use of the proof assistant PVS [20] helped us to avoid unsound shortcuts and to sharpen the arguments.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**, 253–284 (1991)
2. Afek, Y., Brown, G., Merrit, M.: Lazy caching. ACM Trans. Program Lang. Syst. **15**, 182–206 (1993)
3. Dijkstra, E.W.: A class of allocation strategies inducing bounded delays only. Technical Report, Technical University Eindhoven, EWD 319, see www.cs.utexas.edu/users/EWD (1971)
4. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 86, LNCS, vol. 213, Springer, New York, pp. 187–196 (1986)
5. Hesselink, W.H.: Deadlock and fairness in morphisms of transition systems. Theor. Comput. Sci. **59**, 235–257 (1988)
6. Hesselink, W.H.: Eternity variables to simulate specifications. In: Boiten, E.A., Moeller, B. (eds.) MPC 2002, LNCS, vol. 2386, Springer, New York, pp. 117–30 (2002)
7. Hesselink, W.H.: Refinement verification of the lazy caching algorithm. manuscript in preparation, see www.cs.rug.nl/∼wim/pub/mans.html (2006)
8. Hesselink, W.H.: Using eternity variables to specify and prove a serializable database interface. Sci. Comput. Program **51**, 47–85 (2004)
9. Hesselink, W.H.: Eternity variables to prove simulation of specifications. ACM Trans. Comp. Logic **6**, 175–201 (2005)

10. Hesselink, W.H.: Universal extensions to simulate specifications. Manuscript in preparation, see www.cs.rug.nl/ wim/pub/mans.html (2005)
11. Jonnson, B.: Simulations between specifications of distributed systems. In: Baeten, J.C.M., Groote, J.F. (eds.) CONCUR '91, LNCS, vol. 527, Springer, New York, pp. 346–360 (1991)
12. Jonsson, B., Pnueli, A., Rump, C.: Proving refinement using transduction. Distr. Comput. **12**, 129–149 (1999)
13. Ladkin, P., Lamport, L., Olivier, B., Roegel, D.: Lazy caching in TLA. Distr. Comput. **12**, 151–174 (1999)
14. Lamport, L.: A simple approach to specifying concurrent systems. Commun. ACM **32**, 32–45 (1989)
15. Lamport, L.: How to write a proof. Am. Math. Month. **102**, 600–608 (1993)
16. Lamport, L.: The temporal logic of actions. ACM Trans. Program Lang. Syst. **16**, 872–923 (1994)
17. Lynch, N., Vaandrager, F.: Forward and backward simulations. Part I: Untimed systems. Inf. Comput. **121**, 214–233 (1995)
18. Milner, R.: An algebraic definition of simulation between programs. In: Proceedings of the 2nd International Joint Conference on Artificial Intelligence, British Computer Society 1971, pp. 481–489
19. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Inf. **6**, 319–340 (1976)
20. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference, http://pvs.csl.sri.com (2001)